Keyword Search over XML Streams: Addressing Time-Stamping and Understanding Results

Maciej Gawinecki, Federica Mandreoli, Giacomo Cabri
Dipartimento di Ingegneria dell'Informazione
University of Modena and Reggio Emilia
Via Vignolese, 905
41100 Modena, Italy
{maciej.gawinecki, federica.mandreoli, giacomo.cabri}@unimore.it

ABSTRACT

Algorithms evaluating a keyword query over XML document streams have a chance to become a core part of a new class of information filtering systems. However, the problem of creating a common benchmark for such systems has not received much attention in the community yet. We present our initial contribution to create such a benchmark by introducing a method of identifying correct result nodes across streams. We practically prove that our method can be used to understand behaviour of a search heuristic, compare effectiveness of various search heuristics and efficiency of processing algorithms (implementations) employing the same search heuristic.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Selective dissemination of information; K.6.2 [**Management of Computing and Information Systems**]: Benchmarks

General Terms

Performance

Keywords

benchmarking, information filtering system, keyword query, streams, XML

1. INTRODUCTION

There are numerous online data sources, such as podcast hosts, news sites and blogs, which are disseminating their content in the form of XML document *streams* [10]. In this context, *keyword querying* can be well suited for information filtering systems [17, 10] that personalize information delivered to the user with respect to the registered user queries, because the user does not have to know the schema of data source nor she is forced to formulate complicated structural queries (as – for instance – in YFilter [6]).

The ultimate goal of all information filtering systems is to filter all incoming documents (or document fragments) and return only those that are relevant to the registered user profile query. At the same time, processing time of a user query over an arriving document should be as short as possible. Algorithms that evaluates a keyword query over XML document streams [17, 10] have a chance to become a core part of a new class of information filtering systems. However, the problem of creating a common benchmark for such systems has not received much attention in the community yet as happened in the case for approximating structural queries over real-time data (for example Linear Road [1] or NEXMark [13] benchmarks). Such a benchmark is necessary for the following reasons:

- to compare *effectiveness* of search heuristics for the given combination of data and a query,
- to compare existing *algorithms* and *implementations*,
- to give some *guidelines* in developing efficient and effective systems employing keyword queries evaluated over XML document streams.

These will be crucial issues of information filtering in the near future, as we expect more algorithms of a single heuristic together with even more implementations of these algorithms. Our initial contribution to create such a benchmark is the following.

- A brief specification on how performance of this new class of systems can be measured and compared.
- A way of identifying correct result nodes across XML document streams (*Universal Node Reference*), which is usable for obtaining the following information about tested systems: (i) effectiveness of an employed search heuristic for a given combination of data and query; (ii) processing time of stream processing algorithm (implementation); and (iii) positional information of results in the document, easing understanding of the behaviour of a search heuristics.
- Algorithms to obtain this information within a single over a XML document.

This paper is organized as follows. After sketching background (Section 2), Section 3 explains the concepts related

to benchmarking filtering systems based on keyword query over XML streams. Then, we explain our approach based on Universal Node Referencing (Section 4) and how it can be exploited in benchmarking. Finally, before conclusions (Section 6), we report a real scenario to show the application of our approach (Section 5).

2. BACKGROUND

Information filtering systems employing keyword query over XML streams is an arising branch of information filtering systems. According to our knowledge, only two works regarding this topic have been published. Yang and Shi propose the technique based on refining a keyword query to a query graph being a sub-graph of corresponding DTD document [17]. They provide also a stack-based algorithm to evaluating such a query graph over XML document stream using their new search heuristic. Vagena and Moro in their work [10] propose stack-based algorithms for evaluation SLCA [8] and XRank [15] heuristics over XML document streams. None of these works provide a full benchmark of their method. Yang and Shi, as for now, measure only execution time of processing the whole document and memory, with respect to the document size and query length. Vagena and Moro plan to evaluate their solutions in realistic stream environment. Although the ultimate goal of efficient and effective information filtering system is to find all relevant results in the shortest time, the specific nature of systems under consideration makes it impossible to directly re-use existing benchmarks and evaluation methods in similar well tested domains.

Classical information dissemination systems, like SIFT (Stanford Information Filtering System, introducing e-mail based dissemination of USENET news based on keyword user profiles) [16] process and classify *whole documents*, while search heuristic in a keyword query over XML document work on *document fragments*. Let us then compare considered domain with other orthogonal approaches for processing XML data (Figure 1).

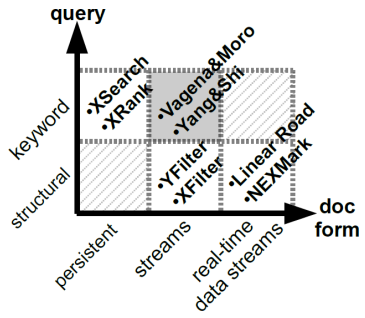

Figure 1: Classes of systems querying XML documents in respect to two dimensions: query type and document representation form and examples of such systems.

Structural query over data streams and real-time data streams. No direct inspiration can be taken also from benchmarks for systems that aggregate and summarize real-time data from streams. In those cases effectiveness (accuracy) and efficiency (timeliness) are combined into a single measure: Tuple Latency, Output Matching (in NEXMark benchmark [13]), and L-Rate (in Linear Road benchmark [1]). In case of keyword queries over streams timeliness is not of concern, because data reported later then expected are still accurate. In systems, which filter streamed XML documents using structural queries (e.g. XFilter, YFilter [6]), there is no need to approximate results as is the case of keyword query systems and thus measure their effectiveness.

Keyword query over persistent data. Regarding efficiency, it is not reasonable to measure query completion time in our case, because continuous querying on streams theoretically never ends [2]. An approach of measuring effectiveness is more likely to be re-used in the considered context, because effectiveness of XML search engines implementing keyword search over persistent XML data is measured in terms of precision of top m results in the ranking (e.g. XSearch [5], XRank [8]) and it directly depends on the underlying search heuristic. In this way, having two XML nodes, each one matching one of the query keywords, two systems employing different heuristics may consider them as relevant if they are closely associated or not. For instance, SLCA[1] [15] heuristic disqualifies a pair of nodes n_1 and n_2 matching query terms if there exists any other matching pair (n_3, n_4) with $LCA(n_3, n_4)$ being a descendant of $LCA(n_1, n_2)$, while XRank [8] heuristic has the additional constraint that $n_1 = n_3$ or $n_2 = n_4$ (LCA is the *Lowest Common Ancestor* [11]). However, it can be a hard task to come from heuristics' definitions to results considered by particular heuristic as relevant. Much insight to this problem brings the work of Vagena at el., which theoretically compares effectiveness of existing heuristics in the case of structurally different XML document fragments [14]. Though this can be still not enough to understand behaviour of a heuristic in practice due to the following issues: (a) new heuristics are introduced, for instance recently Yang and Shi introduced MISLCA [17], a new heuristic that has not been analyzed in work of Vagena et al.; (b) it could be an uphill task to foreseen behaviour a search heuristic on real data, especially for vast volume of data and many query terms; (c) even a single result of a search heuristic can be quite a large document fragment and thus it maybe not easy to understand why it has been chosen.

3. BENCHMARKING KEYWORD QUERY ON XML STREAMS

Popular information filtering systems (YFilter, SIFT) filter on the level of whole documents. YFilter system considers a document as matching to particular user query (XPath expression), if it contains at least one fragment satisfying the query. On the other hand, SIFT employs word frequency approach and thus requires finding all occurrences of keyword in the document to decide whether it is relevant enough [16]. On the contrary, in the context of XML documents, the issue of keyword search reduces to the problem of finding the smallest document fragments containing the XML nodes that satisfy query terms[2]. This has a fundamental impact on the way effectiveness and efficiency of

[1]SLCA – Smallest Lowest Common Ancestor.

[2]Precisely, resulted document fragment can be defined differently, e.g. as (a) a subtree containing all nodes laying on path from the LCA to leaf nodes of the original document, (b) Minimum Connecting Tree connecting all matching nodes together with their [17], or (c) part of the docu-

a system are measured. Therefore, in the context of such systems benchmarking performance can be simply reduced to:

- *Measuring effectiveness of a search heuristic*, because effectiveness of the system can be reduced to effectiveness of employed heuristic (see Section 3.1). Indeed, depending on the underlying XML document structure, heuristics differ in false/negative positive/negatives returned for the given query [14]. There have been devised a number of search heuristics. In this paper we will exemplify our considerations with two of them: SLCA, XRank.
- *Measuring efficiency of an algorithm* (and its implementation) applying a selected heuristic on streams (see Section 3.2).

3.1 Defining effectiveness

In systems in which a *relevant result* is a result returned by an employed heuristic, the issue of measuring system's effectiveness can be reduced to measuring effectiveness of the heuristic. Therefore the following problems must be addressed:

- measuring effectiveness of a particular heuristic for the given combination of data and a query (Section 3.1.1),
- understanding pros and cons of the heuristic on practical base, by understanding results of such a heuristic (Section 3.1.2).

In the following two sections we sketch a solution of these problems.

3.1.1 Measuring heuristic effectiveness

If algorithm applying a particular search heuristic on persistent data exists, then effectiveness of the heuristic can be easily measured on persistent representation of streamed XML document, because it does not depend on the representation form of XML document (streamed or persistent). We take our inspirations from classical persistent information retrieval systems, which employ keyword query, where *real result* is considered as relevant if it belongs to the predefined *relevant answers set*. Relevant answers set is usually automatically constructed from results of a structural query executed over the same set of data. Obviously, the structural query must express the same *user information need* that stands behind the keyword query. A structural query language allows to express this need more explicitly and precisely. Figure 2 illustrates this approach in context of XML document stream. Results of keyword search can be uniquely identified through their root nodes.

EXAMPLE 1. *Let us suppose we are looking in DBLP data presented in Figure 3 for publications written by both Wooldridge and Jennings. The expected answer is the article(2) node. The system we are stressing provides a query language, which*

ment containing nodes matching all query terms without a part belonging to any other result [10].

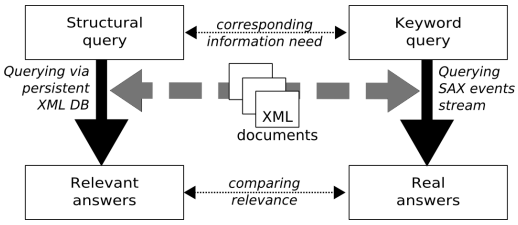

Figure 2: Methodology of estimating effectiveness of keyword query system processing XML document stream.

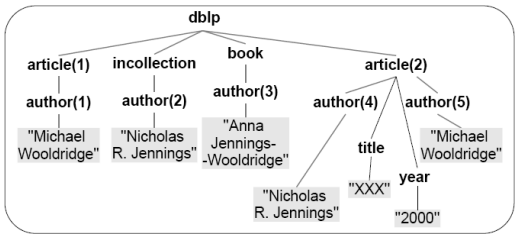

Figure 3: Fictional excerpt of DBLP data.

enables a user to explicitly specify constraints on the labels and/or textual content of a node [14]. Therefore, the same information need could be expressed with <author::Jennings, author::Wooldridge> query. In respect to the definition of both heuristics, their result would be the smallest subtree in XML document, that contains the nodes satisfying all query terms of the query, i.e. at least one author-labelled node containing Jennings word and at least one author-labelled node containg Wooldridge word. Besides returning one true positive heuristics they would return also the following false positives: (a) dblp node, i.e. root of the document, because it is root of the smallest subtree containing matching nodes: author(4) and author(5) (meaningfully related only in respect to XRank heuristic); (b) author(3) node, which is both the node containing Jennings and the node containing Wooldridge (XRank and SLCA heuristic). To automate the process of discovering which results are relevant we construct a relevant answers set the results of the following XQuery expression which corresponds to the user information need (root nodes of the results are defined by RETURN clause):

```
FOR $x IN collection("dblp.dbxml")/dblp/*
  FOR $y IN $x/author
  FOR $z IN $x/author
WHERE
  functx:contains-word($y,"Jennings")
  AND functx:contains-word($z,"Wooldridge")
  AND $y != $z
RETURN $x
```

3.1.2 Understanding search results

During experimenting with particular heuristics, it is not always obvious why the given heuristics returns the given document fragment as a result. Initial knowledge about the node context contained in the node reference can be useful in the same way, as a name of error-prone method is useful during debugging a program. And by the same analogy, since definition of the method gives more information for debugging, thus a user should be able to use the node reference to

query persistent database for the node context (node-rooted subtree). As we will practically prove (Section 5.1), it may be enough to known the name and positional information of a root of the answer and of nodes in the answer that have been found to match query terms.

3.2 Defining efficiency

Two measures are important to describe efficiency of the system under consideration:

- processing time (or its variant, e.g. Multi-Query Processing Time independent to the parser's implementation [6]),
- memory used to store/index user queries and to cache information gathered during processing a stream.

Both measures should be reported together with selectivity of queries, number of registered queries and average number of query terms. We do not claim reporting document sizes, as it is done by Yang and Shi [17], because a system under consideration returns document fragment as results, not whole documents. In this way also different algorithms and implementations employing the same search heuristic can be compared at finer grain.

3.3 Need for node identification

The analysis of problems that must be addressed in designing benchmark for keyword query over XML streams drive us to the conclusion that some method of unique identification of nodes must be introduced to such a benchmark. Practically, we need node identification to:

- Compare real results with relevant answers set in both persistent and mixed streamed results/persistent environments (Section 3.1.1). This requires overcoming differences in both (a) physical representation of node reference in stream and persistent XML processor and (b) range of provided information in both those environments.
- Measure processing time of document fragments (Section 3.2) by identification of nodes arriving in the stream and leaving the system.
- Understand better behaviour of a search heuristic for the given combination of data and the query (Section 3.1.2).

4. UNIVERSAL NODE REFERENCE

Universal Node Reference will be used as a mean of identify nodes for aforementioned purposes. We require it to have the following properties:

1. *Unique* within a certain XML document, so it does not reference (match) more than one node in a certain document.
2. *Uniform*, i.e. independent of the physical representations of XML document used by particular XML processors. In this way it will be also serializable, so it will be possible to store in it a file for further re-use outside of the original XML processor, e.g. to repeat benchmark for different search heuristics for the same relevant answers set.
3. *Possible to construct* from physical representations of the given node. Practically, it must be possible to obtain such a node reference (a) from a result of a query executed over persistent representation and (b) during parsing a stream of SAX events from a processed node.
4. *Possible to get a referenced node back*, i.e. it must be possible to translate the identifier into physical data address and thus obtain corresponding document fragment in a persistent XML database.
5. *Context-rich*, giving some initial knowledge to the benchmark user about the position of a node in the document, e.g. a name of the node and a name of its parental node.

4.1 Proposed solution

Many node labelling schemata have been proposed, including those based on: (a) *tree-traversal order* (e.g. pre-order and post-order [7]), (b) *textual positions* of the start and end tags (e.g. region encoding [4]) and (c) *path expressions* (e.g. Dewey ID [12], PID [3], Extended Dewey ID [9]). Though labelling schemata have been developed to capture structural information of XML document, they appear also to be useful as Universal Node Reference. Practically, we want to re-use the existing ones that satisfy all required properties. Properties 1-3 are satisfied by all labelling schemata and thus will be commented in the next section, property 4 is satisfied by schema based on path expressions, but only PID (*Path Identifier*) has the 5^{th} property.

Precisely, when constructing an identifier, we will use a notion of a *root data path*, which is an early step of constructing/encoding PID, and thus it is easier to read by the user. Root data path is a combination of *root label path* (sequence of labels of nodes laying on the path connecting the root of the document and a corresponding node) and *node positions* (to distinguish a single node among sibling nodes of the same label) [3]. For example the following root data path identifies the *author(5)* node in the Figure 3: $((dblp, article, author), (1, 2, 2))$, or in its alternative representation: `/dblp[1]/article[2]/author[2]`. We will call this representation *Universal Node Reference*. Please note, that such an identifier is a valid XPath expression (a root-to-node path with numeric predicates) and thus can be directly used to query persistent XML database for a subtree rooted at the corresponding node ($\mathbf{4}^{th}$ **property**). Such a reference gives also information about name of the referenced node, names of all its ancestors and positional information about the node and its ancestor among sibling nodes of the same name ($\mathbf{5}^{th}$ **property**).

Measuring processing time of document fragments requires time-stamping of nodes arriving in the stream and leaving the system, as it is done systems evaluating structural queries over streams [2]. Practically, processing time of a node n can be defined as $\Delta t(n) = t_{out}(n) - t_{in}(n)$, where $t_{in}(n)$ and $t_{out}(n)$ return absolute time of arriving and leaving the system by the node n, accordingly. When bench-

marking a system we will report processing time ($\Delta t(n)$), associated with a reference of the processed node n:

$$[UniversalNodeReference(n), \Delta t(n)].$$

4.2 Other Labelling Schemata

By their definition, all labelling schemata uniquely identify a node in the scope of a single document (**1st property**). They are also independent of the physical representation of the document (**2nd** property), unless they are encoded into physical representation inside of XML processor implementation. Labelling schemata are part of index structures used by persistent XML databases and thus they do not occur in stream XML processors (SAX parser directly provides only column and line of start and end tags). Moreover, even in the context of persistent XML databases there is no one commonly used labelling schema. However, it is possible to construct a label of the given node indirectly, by using DOM-like API of any persistent XML database or through the cache of a SAX parser (**3rd property**). Note also, that obtaining a referenced node in persistent XML database (**4th property**) from other labelling schemata is not so straightforward as from PID, with except to those based on path expression (XPath query API of a database can be used). All others requires additional programming effort[3]. Finally, **5th property** is satisfied only by the schemata based on path expressions. Although methods based on tree-traversal order and on textual positions preserve positional information within the hierarchy of XML document, information contained by a single element is very limited [9]. For example having pre-order and post-order ranks of two nodes we can recognize whether they are in ancestor-descendant relationship, however from the rank of a single node we cannot know its ancestor's nor its own name.

4.3 Reference Construction Algorithms

We will now show that a universal node reference can be easily constructed from results of a query executed via both persistent database (Section 4.3.2) and stream (Section 4.3.3) over the same XML document. It can be done by using information about ancestor and sibling nodes of the given node accessible in each of these environments (Section 4.3.1). The only difference is in the way of accessing this data. We will also show how processing time of node in a stream can be measured with usage of Universal Node Reference (Section 5).

4.3.1 Positional information in stream and persistent environments

Table 1 shows that access to positional information about a node, if possible, can be:

- *direct*: in persistent database, via DOM-like API (e.g. `getParent()` method), in SAX parser – via callback methods (URI/label) or SAX Locator (occurence of an event in terms of column and row in the document) or

[3]Practically, by counting a number of nodes from the document's root for schemata based on tree-traversal order or by counting a number of words from the document's root for schemata based on textual position.

Table 1: Access to node positional information in two XML processors. Access can be direct, after additional computation or with a use of parsing cache.

Node information	Persistent DB	SAX Parser
label/URI:	direct	direct
column/row:	not preserved	direct
pre-order rank:	computable	via cache
post-order rank:	computable	via cache
previous sibling:	direct	via cache
next sibling:	direct	unknown
parent/ancestor:	direct	via cache
child/descendant:	direct	via cache
root-to-node path:	computable	via cache

```
1.  getReferencePath(n: Node): String
2.    if not n is nil then
3.      pos := getPosition(n)
4.      path := concat( getReferencePath(n→parent),
5.        "/", n → label, "[", pos, "]")
6.    else
7.      path := ""
8.    endif
9.    return path
10.
11. getPosition(n: Node): Integer
12.   pos := 0
13.   original := n
14.   while (n not is null) do
15.     if n→label = original→label then
16.       pos := pos + 1
17.     endif
18.     n := n → previousSibling
19.   endwhile
20.   return pos
```

Figure 4: Algorithm for obtaining node reference in persistent XML database.

- *computable*: in persistent database, by traversing among sibling, ancestor and descendant nodes via DOM-like API or
- *via cache*: in stream environment, where most of the structural relationships can be discovered, because: (a) parsing is done in respect to in-order and (b) evaluation whether a candidate node is a root of result is done by XRank and SLCA heuristics when all its descendants has been already visited [10].

4.3.2 Obtaining Reference from Persistent Database

Persistent XML database does not have a way to create a generic XPath expression for a specific internal node reference. Specifically, information about the position of a node in collection of siblings is not maintained in index, and thus it cannot be obtained in a straightforward way. However, we assume that persistent XML database provides DOM-like API to traverse among parent and sibling nodes of the given node, as it is done in the popular XML database Berkeley DB XML. An algorithm for constructing universal node reference in a generic persistent database is presented in Figure 4.3.2. Function `getReferencePath()` returns such universal reference for the given internal reference to a node. It creates root-to-node path by recursively traversing among ancestors of the given node (function `concat()` concatenates

```
1.  onDocumentStart(stack: Stack)
2.    node := ∅ {create new node}
3.    node→label := ∅
4.    node→children := ∅ {create new siblings map}
5.    push(stack, node)
6.
7.  onElementStart(stack: Stack, label: String)
8.    siblingsMap := top(stack)→children
9.    pos := getValue(siblingsMap, label) + 1
10.   setValue(siblingsMap, label, pos)
11.   node := ∅ {create new node}
12.   node→inTime := currentTime()
13.   node→label := label
14.   node→children := ∅ {create new siblings map}
15.   push(stack, node)
16.
17. onElementEnd(stack: Stack)
18.   { if recently processed node is matching/result }
19.   unr := getReferencePath(stack)
20.   node := pop(stack)
21.   node→outTime := currentTime()
22.   processingTime := node→outTime-node→inTime
23.   log(unr,processingTime)
24.
25. onDocumentEnd(stack: Stack)
26.   pop(stack)
27.
28. getReferencePath(stack: Stack): String
29.   path := ""
30.   for i from 1 to size(stack)-1
31.     lastLabel := stack[i]→label
32.     siblingsMap := stack[i-1]→children
33.     pos := getValue(siblingsMap,lastLabel)
34.     path := concat(path, "/",
35.       lastLabel, "[", pos , "]" )
36.   endfor
37.   return path
```

Figure 5: Algorithm for obtaining node reference with SAX parser and for time-stamping nodes.

given set of arguments into a single string of characters). For each ancestor, the algorithm computes also its position in the collection sibling nodes of the same label by calling the `getPosition()` function.

4.3.3 Obtaining Reference from Document Stream

Similarly to the persistent database, a reference to a node in document stream can be constructed on the base of information of ancestor and sibling nodes of the given node. However, in the case of streams additional effort must be spent to gather this information while parsing a document stream. This is because an element already processed in the stream cannot be retrieved unless it has been cached.

To store cached information, the algorithm employs the `stack` of nodes. The `stack` represents currently processed node and all its ancestors. A `node`, a single element of the `stack`, has two properties, its `label` and `children`. The latter is a map structure denoting how many nodes – being children of the given node and having a certain label – have already been visited. The map structure provides two access methods: `getValue()`, returning value (number) for the given key (label) and `setValue()`, setting the given value for the given key.

Complete algorithm presenting process of caching information in the `stack` is shown in Figure 5. Let us now describe how it works. SAX parser is an event-based XML processor. While processing a XML document stream it invokes callback methods corresponding to occurrence of the following events: *start of a document*, *start of a node element* (opening tag), *end of a node element* (closing tag) and *end of a document*. Caching necessary information starts when a new document appears in the stream (`onDocumentStream()` function). A new node with no `label` and empty `children` map is pushed to the `stack`. Each time a new child of the top node from the `stack` is being processed the algorithm: (1) updates `children` map of the node and (2) adds to the `stack` a new node corresponding to the child node. A node is taken away from the stack when all its descendants in the document stream have already been visited (function `onElementEnd()`). The node, that has been added during opening of the document, is popped from the stack when the event of the end of the document occurs (function `onDocumentEnd()`).

The information cached in the stack is used to generate an Universal Node Reference corresponding to the node on the top of the `stack`. This is done by traversing nodes from bottom to top of the `stack` and concatenating their labels. Additionally, positional information of the i-th node in the stack is taken from its parental node, placed on $(i-1)$-th position in the `stack`.

The algorithm generates such a reference during execution of `onElementEnd()` method, triggered by the event, when all descendant nodes of the corresponding node have already been visited. This is strictly related to the behaviour of evaluation algorithms of both search heuristics [10]. They both retrieve whether the given subtree contains a set of nodes matching a user query and being *meaningfully related* (in respect to the politics of each of the heuristics). If the tree is the smallest one containing these nodes, i.e. when the root of the tree is the *Lowest Common Ancestor* of all these nodes [11], then the root node is returned as a result of the query.

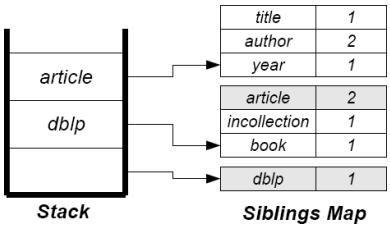

Figure 6: Example of stack of nodes used by the SAX parser.

EXAMPLE 2. *Let us parse the same document (depicted in Figure 3), which has been used in Example 1. Figure 6 shows state of the* `stack` *just after visiting all descendants of article(2). Particularly, it shows it visited the following numbers of nodes in* ***dblp****-rooted tree: 1* ***book*** *node, 1* ***incollection*** *node and 2* ***article*** *nodes. Actually the child of* ***dblp*** *node, 2-nd* ***article*** *node is being processed. Up to this moment the following children for this node has been visited: 1* ***title*** *node, 2* ***author*** *nodes and 1* ***year*** *node. Universal*

node reference generated on the base of the `stack` *would be:* `dblp[1]/article[2]`.

4.3.4 Time-stamping results in stream

The algorithm proposed on Figure 5 reports processing time of a node n together with associated Universal Node Reference (see highlighted lines in the algorithm) with user-defined function `log()`.

5. APPLYING TO A REAL SCENARIO

The of aim the proposed scenario is to show in practice how our proposal can be used to evaluate effectiveness and efficiency and to understand query results of the system processing real XML document streams. According to the methodology presented on Figure 2 we implemented two tools: one evaluating structural query via persistent XML DB , and the other evaluating corresponding keyword query over stream of the same XML document. First tool uses the algorithm presented on Figure 4.3.2 to obtain Universal Node Reference of results found and the other – algorithm on Figure 5). Technically, first tool evaluates a given XQuery expression via Java API of Berkeley DB XML. The other tool employs algorithms proposed by Vagena and Moro for evaluating XRank and SLCA heuristics over XML stream [10] and adopts XRank ranking model [8], simplified for a purpose of usage over highly-structured XML document streams[4].

We experimented with the scenario presented in Example 1 on real DBLP data. The tool for persistent environment was used to generate corresponding relevant answers set. The tool for stream environment was used to evaluate keyword query over the streamed data. In this way we were able to gather the following data just within a single scan over whole DBLP document: (a) *effectiveness* of XRank and SLCA for the given combination of the query and the data, (b) *processing time* of algorithms for evaluation of used heuristics, measured at a document fragment level and (c) *positions of results* found, so it is possible to understand the behaviour of both heuristics for the given combination of the query and the data.

5.1 Understanding results

Figure 7 presents a comparison of relevant answers set with two last results of query evaluation done by the XRank heuristic. Universal Node Reference has been used here not only to identify root nodes of the results, but also for meaningfully related nodes. In this way we are able to answer why the last result has been indicated as non relevant. From the output of comparison process (Figure 7) we can see that root of the answer is the root of whole DBLP document. Moreover, we can also observe, that meaningfully related *author* nodes matching the query terms has been found in two different publications, namely: `/dblp[1]/incollection[4987]` and `/dblp[1]/article[379225]`. It is then obvious that none of the publication nodes can be a relevant itself. This

[4]Please note, that in real stream environment using ranking model makes only sense if we want to rank relevance of a single results without making a whole ranking of all results. This is because querying theoretically never ends and once a result is pushed out by the filtering system to the output stream it gets forgotten.

35.

Rank of result: 1.0	**Processing time (milisec.):** 0.241931

The result is rooted at node:

Is relevant:	**Universal node reference:**
true	/dblp[1]/article[348165]

The result contains the following meaningfully related nodes:

Matching query term:	**Universal node reference:**
author::Jennings	/dblp[1]/article[348165]/author[2]
author::Wooldridge	/dblp[1]/article[348165]/author[3]

36.

Rank of result: 0.4	**Processing time (milisec.):** 221283.822665

The result is rooted at node:

Is relevant:	**Universal node reference:**
false	/dblp[1]

The result contains the following meaningfully related nodes:

Matching query term:	**Universal node reference:**
author::Wooldridge	/dblp[1]/article[379225]/author[3]
author::Jennings	/dblp[1]/incollection[4987]/author[3]

Figure 7: Excerpt of results for <*author::Jennings,author::Wooldridge*> query executed by XRank heuristic over DBLP data.

fact can be easily verified by posing respective XQuery query via Berkeley DB XML shell. For instance, for the *incollection* node the output would be the following:

```
dbxml> query
  collection("dblp.dbxml")
  /dblp[1]/incollection[4987]
1 objects returned for eager expression
  'collection("text.dbxml")
    /dblp[1]/incollection[4987]'
dbxml> print
<incollection mdate="2008-03-06">
  <author>Malka Halgamuge</author>
  <author>Siddeswara Mayura Guru</author>
  <author>Andrew Jennings</author>
  <title>Centralised Strategies for Cluster
    Formation in Sensor Networks.</title>
  <pages>315-331</pages>
  <year>2005</year>
  <booktitle>Classification and Clustering for
    Knowledge Discovery</booktitle>
  ...
</incollection>
```

This is a practical prove of the XRank behaviour described in the work [14]. The XRank heuristic returns a false positive, because the *author* nodes belonging to different publication nodes are *optional elements* (not really meaningful) and the heuristic does not have enough information to prune *dblp* node as an irrelevant result.

5.2 Measuring processing time

On Figure 7 also processing time of resulted document fragment has been shown. This will help in comparing efficiency of benchmarked query evaluation algorithms (and their implementations) with other algorithms (and their implementations) for the same heuristics in the future.

6. CONCLUSIONS

The problem of benchmarking keyword query over XML document streams has not received much attention in the

community yet. In this paper we introduced some directions for benchmarking in this context, proposing a *Universal Node Referencing* approach that allows to uniquely identify nodes in a XML stream. This allows to measure query processing time on the level of document fragment, compare practical effectiveness of search heuristics and understand results of such heuristics by using a single scan over whole XML document. This enables to build benchmarking systems by which the scientific community will be able to compare future proposals. Among many others, we expect algorithms that process several user queries within a single scan of a document fragment, as it is done in those information filtering systems that employ XPath queries evaluated over streams (see for example YFilter system [6]). We are at the beginning of our work, but the chosen direction, being based on good basis, is the right one to our purpose. We plan to test our approach on real streaming data.

7. ACKNOWLEDGMENTS

The authors thank to Giorgio Villani (Information Systems Group at University of Modena and Reggio-Emilia) and Minor Gordon (Computer Laboratory of University of Cambridge) for fruitful discussions; to George Feinberg (Sleepycat Software) and John Snelson (Oracle) for help in debugging Berkeley DB XML database; to Maria Ganzha for comments on early draft of this paper.

8. REFERENCES

[1] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: a Stream Data Management Benchmark. In *VLDB '04: Proceedings of the Thirtieth international conference on Very Large Data Bases*, pages 480–491. VLDB Endowment, 2004.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 2002. ACM.

[3] J.-M. Bremer and M. Gertz. An Efficient XML Node Identification and Indexing Scheme. Technical Report CSE-2003-04, CS Department, University of California at Davis, 2003.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321, New York, NY, USA, 2002. ACM.

[5] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: a semantic search engine for XML. In *VLDB '2003: Proceedings of the 29th international conference on Very Large Data Bases*, pages 45–56. VLDB Endowment, 2003.

[6] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

[7] T. Grust, M. V. Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Syst.*, 29(1):91–131, 2004.

[8] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 16–27, New York, NY, USA, 2003. ACM.

[9] J. Lu, T. W. Ling, C.-Y. Chan, and T. Chen. From region encoding to Extended Dewey: on efficient processing of XML twig pattern matching. In *VLDB '05: Proceedings of the 31st international conference on Very Large Data Bases*, pages 193–204. VLDB Endowment, 2005.

[10] M. M. Moro and Z. Vagena. Semantic Search over XML Document Streams. In *International Workshop on Database Technologies for Handling XML Information on the Web (DATAX)*, 2008.

[11] A. Schmidt, M. L. Kersten, and M. Windhouwer. Querying XML Documents Made Easy: Nearest Concept Queries. In *Proceedings of the 17th International Conference on Data Engineering*, pages 321–329, Washington, DC, USA, 2001. IEEE Computer Society.

[12] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204–215, New York, NY, USA, 2002. ACM.

[13] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. NEXMark – A Benchmark for Queries over Data Streams (DRAFT). Technical report, OGI School of Science & Engineering at OHSU, Septembers 2008.

[14] Z. Vagena, L. S. Colby, F. Özcan, A. Balmin, and Q. Li. On the Effectiveness of Flexible Querying Heuristics for XML Data. In D. Barbosa, A. Bonifati, Z. Bellahsene, E. Hunt, and R. Unland, editors, *XSym*, volume 4704 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2007.

[15] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 527–538, New York, NY, USA, 2005. ACM.

[16] T. W. Yan and H. Garcia-Molina. The SIFT information dissemination system. *ACM Trans. Database Syst.*, 24(4):529–565, 1999.

[17] W. Yang and B. Shi. Schema-Aware Keyword Search over XML Streams. In *CIT '07: Proceedings of the 7th IEEE International Conference on Computer and Information Technology*, pages 29–34, Washington, DC, USA, 2007. IEEE Computer Society.